

## Modularizzazione

- ▶ Quando abbiamo a che fare con un problema complesso spesso lo suddividiamo in problemi più semplici che risolviamo separatamente, per poi combinare insieme le soluzioni dei sottoproblemi al fine di determinare la soluzione del problema di partenza.
- ▶ **Esempio:** La moltiplicazione di numeri con molte cifre viene suddivisa in moltiplicazioni cifra per cifra e i risultati di queste ultime vengono combinate insieme mediante addizioni.
- ▶ Questo procedimento è applicabile anche alla programmazione.
  - ▶ si suddivide un problema complesso in problemi di volta in volta più semplici
  - ▶ una volta individuati (sotto)problemi sufficientemente elementari si risolvono questi ultimi direttamente
  - ▶ si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema di partenza

- ▶ **Approccio top-down**: si parte dall'alto, considerando il problema nella sua interezza e si procede verso il basso per raffinamenti successivi fino a ridurlo ad un insieme di sottoproblemi elementari
- ▶ **Approccio bottom-up**: ci si occupa prima di risolvere singole parti del problema, senza averne necessariamente una visione d'insieme, per poi risalire procedendo per aggiustamenti successivi fino ad ottenere la soluzione globale.
- ▶ I linguaggi di programmazione mettono a disposizione dei meccanismi di **astrazione** che favoriscono un approccio modulare
  - Astrazione sui dati** - il programmatore può definire nuovi tipi di dato specifici per il particolare problema (**tipi di dato astratti**)
    - ▶ collezioni di valori
    - ▶ operazioni con le quali operare su tali valori
  - Astrazione funzionale** - il programmatore può estendere le funzionalità del linguaggio definendo **sottoprogrammi** che risolvono (sotto)problemi specifici.
    - ▶ i sottoprogrammi sono di solito **parametrici**
    - ▶ possono essere (ri)usati alla stessa stregua delle operazioni built-in del linguaggio

## Funzioni

- ▶ In C i sottoprogrammi si realizzano attraverso le **funzioni**.
- ▶ Una funzione può essere vista come una **scatola nera**:

parametri di ingresso  $\longrightarrow$  **F**  $\longrightarrow$  valore calcolato

- risolve un sottoproblema specifico
- attraverso i parametri e il risultato scambia informazioni con il main e con altre funzioni

### Esempio:

$x$   $\longrightarrow$  **abs**  $\longrightarrow$   $|x|$

$x, y$   $\longrightarrow$  **mcd**  $\longrightarrow$   $mcd(x, y)$

$b, e$   $\longrightarrow$  **exp**  $\longrightarrow$   $b^e$

$x_1, \dots, x_n$   $\longrightarrow$  **sum**  $\longrightarrow$   $\sum_{i=1}^n x_i$

**Esempio:** Definizione di `abs` in C

```
int abs(int x)
{
int ris;
if (x<0)
    ris = -x;
else
    ris = x;
return ris; }
```

## ► Uso della funzione

```
main()
{
int x1, x2, z, w;
...
z = abs(x1);
...
printf("%d\n", w + abs(x2));
...
}
```

- ▶ Il linguaggio deve mettere a disposizione strumenti per
  - ▶ **definire** nuove operazioni astratte (funzioni)
  - ▶ **usare** le nuove operazioni definite
- ▶ Distinguiamo due momenti diversi:
  - ▶ la **definizione della funzione**  
definisce il codice che realizza l'operazione astratta
  - ▶ e la **chiamata della funzione**  
corrisponde all'utilizzo della funzione
- ▶ Ad una stessa definizione possono corrispondere diverse chiamate (come  $z = \text{abs}(x1)$  e  $w + \text{abs}(x2)$  nell'esempio precedente).
- ▶ Nella definizione della funzione, il codice fa riferimento agli **argomenti** o **parametri formali** della funzione (nell'esempio  $x$ )  
 $\implies$  un parametro formale non corrisponde ad un valore vero e proprio: è semplicemente un riferimento simbolico (ad un argomento della funzione)

## Esempio:

```
int exp(int base, int esponente)
{
    int ris = 1;
    while (esponente > 0)
    {
        ris = ris * base;
        esponente = esponente - 1;
    }
    return ris;
}
```

- ▶ I **parametri formali** sono base ed esponente

- ▶ Al momento della chiamata, alla funzione vengono forniti i valori degli argomenti, o **parametri attuali**, rispetto ai quali effettuare il calcolo

```
int exp(int base, int esponente)
{...}
```

```
main() {
int b, e, r1, r2;
...
r1 = exp(2,5);
...
scanf("%d %d", &b, &e);
r2 = exp(b, e);
... }
```

- ▶ Prima chiamata `exp(2,5)`
  - ▶ 2 è il parametro attuale corrispondente a **base**
  - ▶ 5 è il parametro attuale corrispondente a **esponente**
- ▶ Seconda chiamata `exp(b,e)`
  - ▶ `b` è il parametro attuale corrispondente a **base**
  - ▶ `e` è il parametro attuale corrispondente a **esponente**

## Funzioni: definizione

Sintassi:

```
intestazione blocco
```

dove

- ▶ `blocco` è il **corpo della funzione**
- ▶ `intestazione` è l'**intestazione della funzione** ed ha la seguente forma:  
`id-tipo identificatore (parametri-formali)`
- ▶ `id-tipo` specifica il **tipo del risultato** calcolato dalla funzione
- ▶ `identificatore` specifica il **nome** della funzione ed è un qualsiasi identificatore C valido
- ▶ `parametri-formali` è una sequenza (eventualmente vuota) di dichiarazioni di parametro (tipo e nome) separate da virgola

**Esempi:** intestazioni di funzione

- ▶ **int** `abs (int x)`  
**int** `MassimoComunDivisore(int a, int b)`
- ▶ **double** `Potenza(double x, double y)`  
**float** `media (int vet[], int lung)`



## Funzioni: chiamata (invocazione, attivazione)

Sintassi:

`identificatore (parametri-attuali)`

- ▶ `identificatore` è il nome della funzione
- ▶ `lista-parametri-attuali` è una lista di **espressioni** separate da virgola
- ▶ i parametri attuali devono corrispondere in **numero** e **tipo** ai parametri formali

**Esempi:** chiamate di funzioni

```
int mcd, x, y1, y2;  
double exp, w, v, z;  
...  
mcd = MassimoComunDivisore(x+1, y1+y2);  
exp = Potenza(z, 3.0);  
...  
exp = Potenza(z, Potenza(v,w));
```

## Semantica (informale) di una chiamata di funzione

- ▶ Dentro il corpo di una funzione **F** compare una chiamata di un'altra funzione **G**
  - ▶ **F** viene detta funzione **chiamante**
  - ▶ **G** viene detta funzione **chiamata**

**Esempio:** nel `main` c'è un assegnamento `x = abs(x)`;  
⇒ `main` è il chiamante, `abs` il chiamato
- ▶ Una chiamata di funzione è un'**espressione**, la cui valutazione avviene come segue:
  - ▶ viene sospesa l'esecuzione di **F** e viene "ceduto il controllo" a **G**, dopo aver opportunamente associato i parametri attuali ai parametri formali (**passaggio dei parametri**, fra poco ...)
  - ▶ vengono eseguite le istruzioni di **G**, a partire dalla prima
  - ▶ l'esecuzione di **G** termina con l'esecuzione di un'istruzione speciale (istruzione `return`) che calcola il risultato della chiamata (è il valore dell'espressione corrispondente alla chiamata)
  - ▶ al termine dell'esecuzione di **G** il controllo ritorna a **F**, che prosegue l'esecuzione a partire dal punto in cui **G** era stata attivata

## Valore di ritorno di una funzione: istruzione `return`

- ▶ **Esempio:** Funzione che restituisce il massimo tra due interi.

```
int max(int m, int n) {  
    if (m >= n)  
        return m;  
    else  
        return n;    }
```

- ▶ Chiamata di `max`, ad esempio da `main`:

```
main() {  
    int i, j, massimo;  
    scanf("%d%d", &i, &j);  
    massimo = max(i,j);  
    printf("massimo = %d\n", massimo);    }
```

- ▶ La funzione `main` tramite i parametri attuali **comunica** alla funzione `max` i valori sui quali calcolare la funzione (il valore delle variabili `i`, `j`).
- ▶ La funzione `max` tramite il valore di ritorno **comunica** il risultato al `main`.

- ▶ Nel corpo **deve** esserci l'istruzione `return espressione;` la cui esecuzione comporta:
  - ▶ il calcolo del valore di `espressione`: questo valore viene restituito al chiamante come risultato dell'esecuzione della funzione
  - ▶ la cessione del controllo alla funzione chiamante

## Osservazioni

- ▶ in `return espressione`, il tipo di `espressione` deve essere lo stesso del tipo del risultato della funzione dichiarato nella definizione
- ▶ l'esecuzione di `return espressione` comporta la **terminazione** dell'esecuzione della funzione

### Esempio:

```
int max(int m, int n) {  
    if (m >= n)  
        return m;  
    else  
        return n;  
    printf("pippo");    /* non viene mai eseguita */ }  
}
```

## Dichiarazioni di funzione (o prototipi)

- ▶ I parametri attuali nella chiamata di una funzione devono corrispondere in numero e tipo (in ordine) ai parametri formali.
- ▶ Dobbiamo permettere al compilatore di fare questo controllo  
⇒ prima della chiamata deve essere nota l'intestazione
- ▶ Due possibilità:
  1. la funzione è stata **definita** prima
  2. la funzione è stata **dichiarata** prima

### Sintassi della **dichiarazione di funzione** (o **prototipo**)

`intestazione;`

ovvero:

`id-tipo identificatore (parametri-formali);`

- ▶ c'è un “;” finale al posto del blocco
- ▶ nella lista di parametri formali può anche mancare il nome dei parametri — interessa solo il tipo
- ▶ il compilatore usa la dichiarazione per controllare che l'attivazione sia corretta
- ▶ dopo **deve** esserci una definizione della funzione coerente con la dichiarazione

## Ordine di dichiarazioni e funzioni

- ▶ Bisogna dichiarare o definire ogni funzione **prima** di usarla (chiamarla)
- ▶ È pratica comune specificare in quest'ordine:
  1. dichiarazioni (**prototipi**) di tutte le funzioni (tranne **main**)
  2. definizione di **main**
  3. definizioni delle funzioni
- ▶ In questo modo ogni funzione è stata dichiarata prima di essere usata
- ▶ L'ordine in cui mettiamo le definizioni non deve necessariamente corrispondere a quello delle dichiarazioni.

## Esempio:

```
int max(int, int);
```

```
int foo(char, int);
```

```
main() ...
```

```
int max(int m, int n) ... /* OK. definizione coerente  
con il prototipo */
```

```
int foo (int z, char c) ... /* NO! definizione non coerente  
con il prototipo */
```

Nella definizione di `foo` i parametri formali non sono nell'ordine specificato dal prototipo.

# Passaggio dei parametri

- ▶ Abbiamo visto che le funzioni utilizzano **parametri**
  - ▶ permettono uno **scambio di dati** tra chiamante e chiamato
  - ▶ nell'intestazione/prototipo: lista di **parametri formali** (con tipo associato) – sono delle variabili
  - ▶ nell'attivazione: lista di **parametri attuali** — possono essere delle espressioni
- ▶ Al momento della chiamata ogni **parametro formale viene inizializzato al valore del corrispondente parametro attuale.**
- ▶ Il valore del parametro attuale viene **copiato** nella locazione di memoria del corrispondente parametro formale.
- ▶ Questo meccanismo di passaggio dei parametri viene comunemente detto **passaggio per valore.**



## Esempio:

```
int succ (int);    /* prototipo di succ */

main()
{  int z, w;
   z = 10;
   w = succ(z);
   printf("%d", w);
}

int succ (int x)
{  x = x + 1;
   return x;
}
```

L'effetto della chiamata `succ(z)` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
x = 10;          /* il parametro formale viene inizializzato con
                  il valore del parametro attuale */
x = x + 1;       /* esecuzione del corpo della funzione
return x;        succ */
```

- ▶ Chiamate diverse corrispondono ad inizializzazioni diverse delle variabili corrispondenti ai parametri formali

```
w = succ(20);      x = 20;
                   ⇒  x = x + 1;
                   return x;
```

- ▶ In questo caso il valore assegnato alla variabile  $w$  è 21.

```
z = 10;
w = succ(z+3);     x = 13;
                   ⇒  x = x + 1;
                   return x;
```

- ▶ In questo caso il valore assegnato alla variabile  $w$  è 14.
  - ▶ Se non vi è corrispondenza perfetta tra il tipo del parametro formale e quello del parametro attuale, viene effettuata una **conversione implicita** di tipo secondo le regole già viste.

# Procedure

- ▶ Non sempre le operazioni astratte di cui abbiamo bisogno possono essere descritte in modo naturale come funzioni matematiche.

**Esempio:** progettare un'interfaccia utente per la stampa di figure geometriche, in cui l'utente può scegliere:

1. la forma della figura
  2. la dimensione
  3. il carattere di riempimento
  4. ...
- ▶ In questo caso il compito dell'operazione astratta non è (o non è soltanto) produrre un valore, ma è produrre effetti di altro tipo, tipicamente **modifiche di stato**.  
⇒ in questi casi possiamo utilizzare **procedure**
    - ▶ le **procedure** sono un'astrazione delle **istruzioni**
    - ▶ le **funzioni** sono un'astrazione degli **operatori**

## Le procedure in C

- ▶ Una procedura è una funzione avente come tipo del risultato il tipo speciale `void`.
- ▶ La definizione/dichiarazione di procedure e la loro chiamata è analoga al caso delle funzioni

### Esempio:

```
void emoticon (int n)
{ /* stampa n volte la sequenza -:) */
  int i;
  for (i=0; i<n; i++)
    {putchar('-'); putchar(':'); putchar(')'); putchar(' ');}
}

main() {
  ...;
  emoticon(3);
  ... }
```

- ▶ Le procedure non contengono di solito un'istruzione `return` (se la contengono è del tipo `return;` che non comporta il calcolo di alcun valore, ma solo la cessione del controllo al chiamante)

- ▶ La semantica di una chiamata di procedura **P** da una funzione/procedura **F** è analoga a quella della chiamata di funzione, ma una chiamata di procedura è un'istruzione
- ▶ In particolare, il passaggio dei parametri avviene per **valore** come nel caso delle funzioni
- ▶ il controllo viene restituito al chiamante al termine dell'esecuzione del blocco che costituisce il corpo della procedura (o in corrispondenza dell'esecuzione di un'istruzione del tipo **return;**)
- ▶ Il C non distingue tra funzioni e procedure (queste ultime sono casi particolari di funzioni)  
⇒ concettualmente, però, è bene vedere le funzioni come astrazioni di **operazioni** e le procedure come astrazioni di **istruzioni**.

## Esempio:

Procedura che stampa una cornice di asterischi di “altezza” parametrica

```
void stampaCornice(int altezza)
{
    int i;
    printf("*****\n");
    for (i=1; i<=altezza; i++)
        printf("          *\n");
    printf("*****\n");
    return;
}
```

- ▶ Come astrazione delle istruzioni, le procedure possono dover **modificare lo stato**.

**Esempio:** Procedura `abs` che **assegna** ad una variabile intera il suo valore assoluto

- ▶ il chiamante deve comunicare alla procedura la variabile in questione
- ▶ la procedura deve analizzare il valore della variabile e, se necessario, effettuare il rimpiazzamento

- ▶ La seguente realizzazione della procedura non è corretta

```
void abs(int x)
{   if (x < 0)
        x = -x;   }
```

- ▶ Simuliamo il comportamento di una chiamata della procedura (come visto in precedenza)

```
int z = -5;
abs(z);            $\implies$    x = -5;
                    if (x < 0) x = -x;
```

- ▶ La modifica del parametro formale **non** si ripercuote sul parametro attuale (si ricordi che il passaggio dei parametri è **per valore**).

## Variabili locali

- ▶ Il blocco che costituisce il corpo di una funzione/procedura può contenere dichiarazioni di variabili.

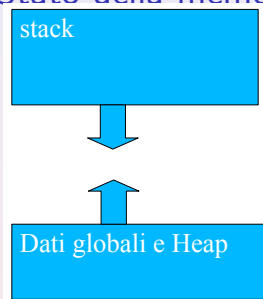
### Esempio:

```
void leggiVettore(int v[], int dim)
{
    int i;          /* i E' UNA VARIABILE LOCALE */
    for (i = 0; i < dim; i++) { ... }
}
```

- ▶ sono variabili proprie della funzione
- ▶ hanno **tempo di vita** limitato alla durata della chiamata
- ▶ più in generale: un identificatore dichiarato nel corpo di una funzione è detto **locale** alla funzione e **non è visibile all'esterno** della funzione (ad esempio nel `main`), ma solo nel corpo della stessa
- ▶ In realtà, ciò non è altro che un **caso particolare** di regole generali che governano la **visibilità** e il **tempo di vita** degli identificatori di un programma.



## Stato della memoria



### Struttura generale di un programma C

- ▶ parte direttiva
- ▶ parte dichiarativa **globale** che comprende:
  - ▶ dichiarazioni di costanti
  - ▶ dichiarazioni di tipi (li vedremo ...)
  - ▶ dichiarazioni di variabili (**variabili globali**)
  - ▶ prototipi di funzioni/procedure
- ▶ il programma principale (**main**)
- ▶ le definizioni di funzioni/procedure

# Esempio

```
#include <stdio.h>          /* parte direttiva */
#define LUNG 10

int i = 1;                  /* variabili globali */
int j = 2;

int Q(int);                /* prototipi di funzioni e procedure */
void P(int *);

main()                      /* programma principale */
{
  int x = 10;
  char c = 'a';
  x = Q(x);
  P(&x);
}

int Q(int v) { ... }      /* definizioni di funzioni e procedure */
void P(int *z) { ... }
```

# Blocchi

- ▶ il corpo di una funzione/procedura, così come il corpo del programma principale, è un **blocco**.
- ▶ In C un blocco è costituito da
  - ▶ una parte dichiarativa (può non esserci)
  - ▶ una parte esecutiva (sequenza di istruzioni)
- ▶ Nel **main** o nel corpo delle funzioni possono comparire diversi blocchi, che possono essere
  - ▶ **annidati**: un blocco è una delle istruzioni di un altro blocco
  - ▶ **paralleli**: blocchi che fanno parte della medesima sequenza di istruzioni

```

{
    int x;
    x = 10;
    {
        int z;
        z = 20 ;
        ...
    }
    ...
}

{
    int x;
    x = 10;
    ...
}
{
    int z;
    z = 20;
    ...
}

```

- ▶ Anche la parte esecutiva del programma principale e di una funzione/procedura è un blocco
- ▶ Gli identificatori dichiarati nella parte dichiarativa di un blocco sono detti **nomi locali** del blocco e devono essere tutti **diversi** tra loro
  - ▶ nel caso di una funzione/procedura, fanno parte dei nomi locali anche gli identificatori utilizzati per i parametri formali

### Esempio:

```
{  
int x; /* NO! identificatore x dichiarato */  
char x; /* due volte nello stesso blocco */  
...  
}
```

```
void p(int x, char y)  
{  
int x; /* NO! identificatore x già' usato per un parametro formale */  
...  
}
```

- ▶ In blocchi diversi possono essere utilizzati gli stessi identificatori

## Esempio:

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
  {
    char x;  /* x: variabile locale del blocco annidato */
    ...
  }
...
}

void p(int x)
{
int y;      /*x,y: variabili locali della procedura p */
...
}
```

- ▶ Un programma C può avere una struttura molto complessa a seguito dell'uso di funzioni, procedure e blocchi.
- ▶ È necessario definire regole precise per regolamentare l'uso dei nomi utilizzati all'interno di un programma.
- ▶ A questo scopo introduciamo alcune definizioni utili.
  - Ambiente globale:** è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa globale del programma
  - Ambiente locale di una funzione:** è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa della funzione e nella sua intestazione
  - Ambiente locale di un blocco:** è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa del blocco
- ▶ Quanto detto informalmente in precedenza può essere meglio precisato:
  - ⇒ è possibile dichiarare più volte lo stesso identificatore (anche con significati diversi) purché in ambienti diversi
- ▶ Se ciò evita il proliferare di identificatori, causa il problema di stabilire il significato di un riferimento ad un identificatore in un generico punto del programma

## Esempio: Riprendiamo l'esempio precedente

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
  {
    char x;  /* x: variabile locale del blocco annidato */
    ...
  }
...
}
```

```
void p(int x)
{
int y;      /*x,y: variabili locali della procedura p */
...
}
```

- ▶ Se in un punto del programma viene eseguita l'istruzione `x = ...`, a quale delle **tre** dichiarazioni di `x` ci si riferisce?
- ▶ Dipende dal punto in cui si trova tale assegnamento e dalle **regole di visibilità** (o regole di **scoping**).

## Regole di visibilità

- ▶ Gli identificatori presenti nell'ambiente **globale** sono visibili in tutte le funzioni e in tutti i blocchi del programma.  
Se un identificatore è definito in più punti (in blocchi e/o funzioni), la definizione valida è quella dell'ambiente più vicino al punto di utilizzo.  
**N.B.** Gli identificatori predefiniti del linguaggio si intendono parte dell'ambiente globale.
- ▶ Gli identificatori presenti nell'ambiente **locale di una funzione** sono visibili nel corpo della funzione (ivi compresi eventuali blocchi in esso contenuti).  
Se un identificatore è definito in più punti del corpo, la definizione valida è quella dell'ambiente più vicino al punto di utilizzo.
- ▶ Gli identificatori presenti nell'ambiente **locale di un blocco** sono visibili nella parte esecutiva del blocco (ivi compresi eventuali blocchi in essa contenuti).  
Se un identificatore è definito in più punti di un blocco, la definizione valida è quella dell'ambiente più vicino al punto di utilizzo.



- ▶ Detto altrimenti, l'ambito di visibilità di un identificatore è determinato dalla posizione della sua dichiarazione:
  - ▶ gli identificatori dichiarati all'interno di un blocco hanno ambito di visibilità a livello di blocco
    - ⇒ una variabile dichiarata in un **blocco** è visibile **solo in quel blocco** (compresi eventuali blocchi annidati)
  - ▶ gli identificatori dichiarati all'interno di una **funzione** (compresi quelli nell'intestazione) hanno ambito di visibilità **a livello di funzione**
    - ⇒ una variabile dichiarata in una **funzione** è visibile **solo nel corpo della funzione** (compresi eventuali blocchi annidati)
  - ▶ gli identificatori dichiarati all'esterno delle funzioni e del main hanno ambito di visibilità a livello di programma
    - ⇒ una variabile **globale** è visibile **ovunque** nel programma

## Esempio:

```
int x1=10, x2=20;
char c='a';

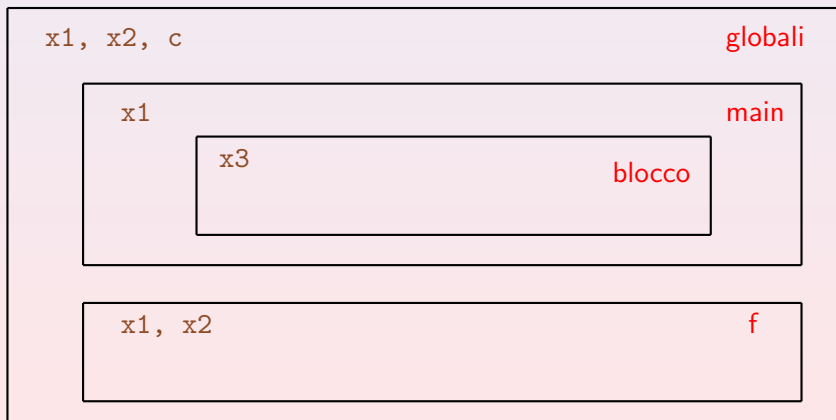
int f(int);

main()
{
int x1=30;    /* nasconde la variabile globale x1 */
x2 = x1+x2;  /* x1 e' quella locale, x2 e' globale */
printf("x1=%d  x2=%d\n", x1, x2);  /* stampa x1=30  x2=50 */
  { int x3=50;
    x1=f(x3); /* x1 e' quella locale al primo blocco */
    printf("x1=%d  x2=%d\n", x1, x2);  /* stampa x1=150  x2=50 */
  }
}

int f(int x1) /* nasconde la variabile globale x1 */
{ int x2;    /* nasconde la variabile globale x2 */
  x2 = x1 + 100; /* x1 e' il parametro formale, x2 la var. locale */
  return x2;
}
```

## Rappresentazione Grafica: Modello a contorni

- ▶ Si rappresenta ogni **ambiente** mediante un rettangolo con gli identificatori in esso contenuti.



# Durata delle variabili

- ▶ Una variabile ha un suo **tempo di vita**.
  - viene **creata** (ovvero ad essa viene riservata uno spazio di memoria)
  - viene (o può essere) **distrutta** (ovvero viene rilasciato il corrispondente spazio di memoria).
- ▶ Si distinguono due classi di variabili:
  - ▶ variabili **automatiche**: vengono create ogni volta che si entra nel loro ambiente di visibilità e vengono distrutte all'uscita di tale ambiente
    - ▶ es. variabili **locali di un blocco**: vengono create all'ingresso del blocco {  
distrutte all'uscita dal blocco }
    - ▶ es. variabili **locali di una funzione**: vengono create al momento della chiamata e distrutte all'uscita
  - ▶ variabili **statiche**: vengono create una sola volta e vengono distrutte solo al termine dell'esecuzione del programma (non ne faremo uso ...)
- ▶ **N.B.** nel caso di funzioni/blocchi eseguiti più volte (es. funzione chiamata in punti diversi, blocco all'interno di un ciclo):  
le variabili automatiche corrispondenti possono essere associate di volta in volta a locazioni di memoria diverse, quindi  
il loro valore **non persiste** tra una esecuzione e la successiva

## Gestione della memoria a tempo di esecuzione (run-time)

- ▶ Il codice macchina e i dati risiedono entrambi in memoria, ma in zone separate:
  - ▶ la memoria per il codice macchina è fissata a tempo di compilazione
  - ▶ la memoria per i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a **pila**
- ▶ Una **pila** (o **stack**) è una struttura dati con accesso **LIFO**: **Last In First Out** = l'ultimo entrato è il primo ad uscire (es.: pila di piatti da lavare).
- ▶ Il sistema gestisce in memoria la **pila dei record di attivazione (RDA)**
  - ▶ per ogni **chiamata di funzione** viene creato un nuovo **RDA** in cima alla pila
  - ▶ al termine della chiamata della funzione il **RDA** viene rimosso dalla pila
- ▶ Ogni **RDA** contiene:
  - ▶ le locazioni di memoria per i parametri formali (se presenti)
  - ▶ le locazioni di memoria per le variabili locali (se presenti)
  - ▶ altre informazioni che non analizziamo
- ▶ Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

## Esempio:

```
int f(int);
main()
{
    int x, y, z;
    x=10;
    y=20;           /* blocco principale */
    z = f(x);       /* prima chiamata di f */
    {
        int x=50;   /* uscita da f e ingresso nel blocco annidato*/
        y=f(x);     /* seconda chiamata di f */
        z=y;        /* uscita da f */
    }
    ...           /* uscita dal blocco */
}
int f(int a)
{
    int z;
    z = a + 1;
    return z;
}
```

[◀ PUNTO 1](#)[◀ PUNTO 2](#)[◀ PUNTO 3](#)[◀ PUNTO 4](#)[◀ PUNTO 5](#)[◀ PUNTO 6](#)

# Evoluzione della pila

|   |    |
|---|----|
| x | 10 |
| y | 20 |
| z | ?  |

▶ PUNTO 1

# Evoluzione della pila

|   |    |
|---|----|
| a | 10 |
| z | ?  |

|   |    |
|---|----|
| x | 10 |
| y | 20 |
| z | ?  |

▶ PUNTO 2



# Evoluzione della pila

|   |    |
|---|----|
| x | 50 |
|---|----|

|   |    |
|---|----|
| x | 10 |
| y | 20 |
| z | 11 |

▶ PUNTO 3

# Evoluzione della pila

|   |    |
|---|----|
| a | 50 |
| z | ?  |
| x | 50 |
| x | 10 |
| y | 20 |
| z | 11 |

► PUNTO 4

# Evoluzione della pila

|   |    |
|---|----|
| x | 50 |
| x | 10 |
| y | 51 |
| z | 11 |

► PUNTO 5

# Evoluzione della pila

|   |    |
|---|----|
| x | 10 |
| y | 51 |
| z | 51 |

► PUNTO 6

## Variabili statiche: un esempio d'uso

- ▶ Una variabile **statica**, una volta creata, rimane in vita per tutto il tempo di esecuzione del programma.

**Esempio:** `f(void) { static int x; ... }`

- ▶ la variabile viene inizializzata alla prima attivazione della funzione
- ▶ conserva il suo valore tra attivazioni successive
- ▶ è locale, quindi visibile solo all'interno della funzione in cui è dichiarata

**Esempio:** Funzione che ritorna il numero di volte che è stata attivata.

```
int fun1(void) {
    static int conta = 0;
    /* variabile locale statica visibile solo in fun1;
       contatore del numero di attivazioni di fun1 */
    .....
    conta++;
    return conta;
}
```